(U//FOUO) Tor

2006 CES Summer Program

Abstract

(U) Tor is an open-source anonymization program created by Roger Dingledine, Nick Mathewson, and Paul Syverson. Originally sponsored by the US Naval Research Laboratory, and now backed by the Electronic Frontier Foundation, it routes a user's traffic through several servers in a way that hides the user's location.

(TS//SI) We have seen several targets using Tor. Our goal was to analyze Tor source code and determine any vulnerabilities in the system. We set up an internal Tor network to analyze Tor traffic, in the hopes of discovering ways to passively identify it. We also worked to create a custom Tor client which allows the user finer control.

Contents

TOP SECRET//COMINT//20291123

1 (U) Tor overview

(U) Tor is an implementation of an onion routing system. Its creators use a special jargon to describe its workings; please consult appendix A, **Tor glossary**, for definitions. Tor's anonymization works by having a Tor client send its traffic through a circuit of three servers, each running Tor, under several layers of encryption. This accomplishes several things. Most basically, the Tor servers, many of which are listed on publicly advertised directory servers, are chosen to act as a series of proxies. This may seem to be excessively complex, as a single proxy server can be used to hide one's location, but a single-hop proxy is vulnerable in two ways. First, by analyzing the pattern of the traffic going to and from the proxy server, it is possible to deduce which clients are making which requests. Second, if an attacker owns the proxy server, then it certainly knows who is asking for what, and anonymization is ruined. By using multiple hops, Tor is much more resistant to both of these attacks. Traffic analysis becomes extraordinarily difficult, as it must be coordinated across several machines, and an attacker must own all the hops along the circuit in order to trace requests back to the originating client.

(U) If Tor's only feature was the use multiple proxies, it would not necessarily mean that it could not be attacked some way, such as knowing which circuit a client will choose to route traffic through, or simply reading requests sent in the clear and looking for identifying information. However, Tor leverages the OpenSSL cryptographic libraries to encrypt the data it sends and to choose random circuits in order to prevent an attacker from gaining useful information.

(U) The series of Tor servers used to relay a client's TCP traffic is referred to as a circuit. Circuits are set up one server at a time in a way that keeps any single server from knowing the entire layout of the circuit. The client connects to the first server. It then asks the first server to connect to the second server. Finally, it sends a request through the first server that asks the second server connect to the third server. Each server only knows who connected to it and to whom it connected, which is just one server each way in the circuit. A more detailed explanation can be found in appendix B.1.

(U) The phrase "onion routing" refers to the way that encryption is layered to hide information sent down the circuit. When setting up the circuit, the client negotiates keys with each of the servers it will be routing through. Most circuits in Tor contain three servers, so the client negotiates three secret keys, one with each server. When the client wants to send some data, for example an HTTP GET request, it first encrypts the data once using each key. It sends the packet to the first server, which removes one layer of encryption and passes along the packet to the second server. The second server removes another layer of encryption and sends the packet to the third server. When the third server removes a layer of encryption, then the packet is now in the clear, and the server will send the HTTP GET to the appropriate destination. In this way, only the third server can see

what the client is sending, and to anyone outside of the Tor cloud it appears that the third node originated the request.

(U) In short, Tor uses a combination of encryption and multiple proxies to hide who is sending what through the Tor cloud.

2 (U//FOUO) Objectives

1. (U//FOUO) Become familiar with Tor by setting up an internal Tor network.
2. (U//FOUO) Read the documentation and code of, as well as examine the packets sent and received by, the client.
3. (S//SI) Develop detailed client specifications and requirements and document them for the community.
4. (S//SI) Create Tor client libraries to be leveraged by a network-enabled application.
5. (S//SI) Create a Tor client, using the above Tor libraries, that allows for on-the-fly tweaking of the client's behavior in the Tor cloud.

(U//FOUO) We feel that we accomplished all of these objectives, as well as a few other unspecified goals. This paper discusses our progress on all of these objectives.

3 (U//FOUO) Objectives 1 and 2

(S//SI) We set up our own internal Tor network on eight machines in the Protocol Exploitation lab. The network consisted of two directory servers, five servers, and 1 machine that was configured to act only as a client. They were connected to each other through the same hub, and we used Ethereal to sniff packets that they sent back and forth.

(U//FOUO) Tor is a large open-source project, and we were able to download its source code and documentation from tor.eff.org. Most of our analysis was performed using source code from Tor versions 0.1.1.17 and 0.1.1.21. There were no major differences between the two versions, and the core functionality of building circuits and maintaining inter-server connections remained unchanged.

(U//FOUO) Our analysis was three-fold: we read the protocol specification, examined the source code, and observed the actual behavior of our own Tor cloud. This gave us an excellent idea of how Tor works, allowing us to formulate hypotheses regarding Tor's behavior which we could then immediately test.

4 (U//FOUO) MJOLNIR

(TS//SI) Another major objective was the specification and development of a custom Tor client library. We developed a library that allows the programmer control over all aspects

of building a circuit and sending data. Our library is as portable as Tor itself, and we have successfully compiled and tested it on both Windows and Linux. We named the library MJOLNIR, which is the name of Thor's hammer from Norse mythology.

(TS//SI) MJOLNIR is a modification of Tor, and it is ideally indistinguishable from an original Tor client. As such, it should appear identical to Tor in traffic. To ease this process, we used original Tor functions whenever possible. However, its main purpose is to provide the programmer with greater control over all aspects of Tor. In the normal Tor client, almost all servers in all circuits chosen randomly. Using MJOLNIR, the programmer can build circuits one server at a time, with no limit to the number of servers in the circuit. Then the programmer can send an arbitrary TCP payload across the circuit that he built and process the response however he wishes.

(TS//SI) All of MJOLNIR's most important calls (in setting up a circuit as well as sending data) are blocking. This is because the Tor code is inextricably linked with libevent, an open-source library that allows the user to set up signals that can be handled whenever they are needed. Whereas Tor uses events and a function called once per second to perform all its necessary checks and corrections, MJOLNIR uses libevent internally so that each function does what it should do before returning control to the calling program. It would be unfair to any programmer to require that any program using MJOLNIR should also be forced to use libevent at the same level that Tor does. We felt that having blocking calls was the lesser of two evils in this situation, as only programs that follow Tor's structure could possibly find libevent a viable way of controlling program flow. MJOLNIR's solution of using libevent internally does make the program slower, but allows the programmer to know whether all of his requests succeed or fail immediately upon return to his program.

4.1 (U//FOUO) Building circuits

(TS//SI) The most fundamental portion of MJOLNIR is its capability to build arbitrary circuits of any length. Using our library, a programmer can build a circuit, use it for some traffic, then extend it and continue using it. Like Tor, MJOLNIR allows multiple circuits to be active at one time, and the user can send traffic along any of them.

(TS//SI) In order to build a circuit, the client must have information about some of the servers in the Tor cloud. This information is kept on the directory servers and their mirrors, and to get this information the client must request it specifically from the directory. MJOLNIR provides a function, `update_router_descriptors`, which automates this process. Once MJOLNIR has the directory, it can build circuits to any of the servers that it knows about. Servers that have middleman-only exit policies are still allowed to be at the end of the circuit as it is being built. However, should the calling program want data to travel across the circuit, it must be using an exit node that will allow traffic to exit to where the program wants it to go, or MJOLNIR will refuse to send the traffic in the first place.

(TS//SI) In order to build a circuit, the programmer must begin with a call to MJOLNIR's function `new_arbitrary_circuit_by_nickname`, which returns a new **circuit_t** structure with the given server as its first node. MJOLNIR also provides a set

of functions that will append another server to a given circuit. There is also a function that will return a random circuit of a given length, which is provided both in order to retain some of Tor's original functionality and to ease the process of constructing a circuit. MJOLNIR is designed to provide as much flexibility as possible, providing the programmer with a wide array of options for building and extending circuits.

(TS//SI) In an effort to provide a robust suite of functions, MJOLNIR provides error-checking at many different levels of circuit construction. Tor servers will fail when asked to connect to themselves, but that error will not become apparent until data is sent across the circuit. To avoid this problem, MJOLNIR will watch what servers are added to the circuit, and it will refuse to add a server if it is currently at the end of the circuit. In this case, the circuit is not destroyed, but there is at least one situation in which attempting to add a circuit will result in the destruction of the entire circuit. If the program attempts to connect to a server that has gone down since being added, then the circuit will be destroyed. This is initiated by the end of the circuit rather than the client, so we have no control over it. The calling program should always check return values to ensure that circuit constructions and extensions were successful.

4.2 (U//FOUO) Sending and receiving data

(TS//SI) MJOLNIR would not be useful if it could not communicate with other computers. Fortunately, Tor makes it easy for MJOLNIR to send and receive data without relying on an outside program. To send data, MJOLNIR must have three things: a message, a destination, and a circuit across which to send the message. Presumably, the first two are application dependent, while the circuit is easy enough to build using MJOLNIR's functions. Once a circuit is built and a message is ready to be sent, the programmer needs to call `send_payload_down_circ` with the correct arguments, and the message should get to the destination. Each new payload creates a new application proxy (AP) connection, which is only used internally. Normally, outside applications (e.g. a web browser) send their messages across Tor using AP connections. Instead of creating AP connections when it gets a request from another program, MJOLNIR creates an AP connection when it has something to send across a circuit.

(TS//SI) Before the message is sent, MJOLNIR sets up libevent events so that it can successfully read and write across that connection. It uses the write event immediately when it sends a payload. It must still listen for a response before returning control to the calling program. In order to provide a reasonable amount of time to receive a response, MJOLNIR uses a timeout scheme modeled on TCP timeouts. If MJOLNIR does not receive a response before the timeout runs down, then it will shut down the connections it is waiting on and return to the calling program.

(TS//SI) Theoretically, there should be a way to shut down connections more elegantly than using timeouts. For example, consider if MJOLNIR shut down a connection when it received a cell that was less than the maximum length. Initially, this seems like a good idea because Tor servers should minimize the number of packets they send by sending maximal-length payloads whenever possible. However, upon inspection of the packets,

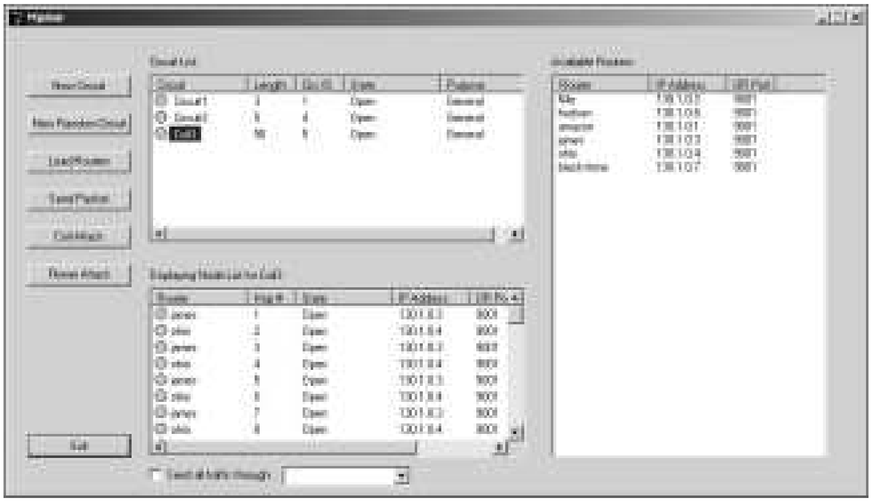


Figure 1: **(TS//SI) The basic GUI window showing (clockwise from top left) circuits, available servers, and circuit contents.**

servers do not always send maximal-length cells, so this method does not work. Also, as far as Tor is concerned, there is no difference between a cell at the beginning of a message and a cell at the end, so there is no easy way to know that a message is complete without parsing the payload.

(TS//SI) Once MJOLNIR receives a cell, it calls a user-set function, `payload_read_handler`. This is a pointer to a function that takes a string (the payload) and its length. Unfortunately, due to the problems described above, there is no way to handle an entire response if it comes in more than one cell; each cell is handled individually. However, by setting this variable (see the API in the appendix), the calling program can do any operation on the incoming cells, thereby parsing out any necessary data. The flexibility afforded by this arrangement is largely what makes MJOLNIR's communication capabilities so useful.

4.3 (U//FOUO) MJOLNIR GUI for Windows

(TS//SI) In addition to porting the MJOLNIR library to Windows in a dynamic link library (DLL), we created a graphical user interface (GUI) to implement a custom Tor client for Windows.

(TS//SI) In its current state, the Windows version of MJOLNIR has the following capabilities:

- Creating new circuits.
- Appending a node to an already existing circuit.
- Creating random circuits of arbitrary length.

- Performing "coil" and "flower" attacks on servers.
- Appending "coils" and "flowers" to existing circuits.
- Sending arbitrary packets through a circuit.

(U//FOUO) See appendix F for detailed information on the Windows version of MJOLNIR.

5 (S//SI) Tor's X.509 certificates

(S//SI) The only part of a normal Tor interaction that does not take place under TLS encryption is the TLS handshake, which has to be done in the clear. We chose to focus on this traffic to see if there was any way to identify Tor traffic, since all traffic under TLS encryption looks the same.

(U) Tor creates two X.509 certificates shortly after starting up, in the function `tor_tls_context_new`. The first certificate contains a short-term connection key, and is signed by the machine's public RSA key (its identity key). The second is a self-signed certificate containing the identity key.

(U) A typical Tor X.509 certificate is shown in table 1.

(S//SI) The string "TOR" appears in the *organizationName* of both the subject and issuer portions of the X.509 certificates. This means that we see this string in the clear during the TLS handshake. By also taking advantage of the ASN.1 encoding that surrounds the string in the certificate, we were able to find a byte sequence that appears in every Tor TLS handshake. This makes collection easy, as we simply look for the byte sequence in traffic. See table 2 for the sequences.

5.1 (S//SI) MJOLNIR's X.509 certificates

(TS//SI) When a Tor server receives a certificate, it will verify that its peer sent it a certificate with all of the required fields for X.509 certificates and that its subject's *commonName* field exists and is alphanumeric. Tor servers will not reject a certificate with a period in them, but they will generate a log message to the effect that someone who is not using Tor is trying to connect to the machine. It does not check that the *organizationName* for either subject or issuer is "TOR," and it does not check to see that the "<identity>" string is ever present. Because of this, MJOLNIR can easily send a certificate with false or random information and still connect to the Tor cloud.

(TS//SI) To send a falsified certificate, MJOLNIR just needs an array of field/value pairs, where the field is the field name of the X.509 certificate (e.g. *commonName*, *organizationName*, *countryCode*), and the value is the corresponding value of the field (e.g. "mjolnir", "pantheon.se", "SE"). As long as the field/value pair is valid, MJOLNIR will put it into the certificate, so that anyone watching traffic will not immediately realize that the traffic from a machine using MJOLNIR is Tor traffic.

Table 1: **(U) Tor's X.509 certificate**

- **Version: 3 (0x2)** This is hard-coded into Tor.
- **Serial Number:** ***some number*** Currently, we don't know if this is a random number, or if there is some significance. (We do know that it is created with an OpenSSL function.)
- **SignatureAlgorithm: sha1WithRSAEncryption**
- **Issuer: O=TOR CN=*string* <identity>** "TOR" will always be present as the *organizationName*. The *commonName* field is more variable. *string* will be the nickname of the server as listed in the public directories. For clients without nicknames, it will be the string "client". Note that it will always be followed by " <identity>".
- **Validity:**
  - **Not Before:** ***time*** This is the time that the TLS certificate was created by TOR.
  - **Not After:** ***time*** By default, this is 2 hours later.
- **Subject: O=TOR CN=*string*** Again, "TOR" is the *organizationName*. This is hard-coded, and will always be present. *string* will be just the nickname of the server in the first certificate, and will contain " <identity>" in the second, self-signed certificate.
  - **Subject Public Key Info:**
    - **Public Key Algorithm: rsaEncryption**
      - **RSA Public Key:**
      - **Modulus (1024-bit):** ***some number*** Varies.
      - **Exponent: 65537(0x10001)** This is hard coded into Tor.
- **Signature Algorithm: sha1WithRSAEncryption** This is followed by 1024 bits of signature.

- Byte sequence found in all Tor X.509 certificates:
  `30 xx 31 0c 30 0a 06 03 55 04 0a 13 03 54 4f 52 31 xx 30 xx 06 03 55 04 03`
- Byte sequence found in X.509 certificates of clients with default settings:
  `30 2a 31 0c 30 0a 06 03 55 04 0a 13 03 54 4f 52 31 1a 30 18 06 03 55 04 03 14 11 63 6c 69 65 6e 74 20 3c 69 64 65 6e 74 69 74 79 3e`
  This corresponds roughly to the part of the certificate that says organizationName=TOR and commonName=client <identity>.

Table 2: **(S//SI) Tor byte sequences**

(TS//SI) If the programmer does not provide MJOLNIR with the subject's *commonName*, then it will randomly generate (using OpenSSL's random byte generator) a legal *commonName* between 10 and 21 bytes long. This ensures that even if the programmer forgets to give a certificate to the program, there will be no identifying information in the TLS handshake.

6 (U) Tor's hidden services

(U) Tor allows machines to offer a TCP service without revealing their IP address, using the same anonymization techniques as clients connected to the network. Such services can only be connected to through the Tor network, and are accessed by special `.onion` addresses resolved by Tor directory servers. This offers several benefits to the operator of the service. Such a service can be hosted from behind a firewall or NAT, and any attackers attempting to perform a distributed-denial-of-service attack will be stymied because they do not know the machine's IP address. For details on the hidden service protocols, see appendix C.

(S//SI) Currently, there are roughly 50 publicly advertised hidden services. This number seems low. Given Tor's rising popularity it seems likely that more than 50 hidden services exist. The ability to locate hidden services is important, as only then will we be able to police and/or exploit them. Unfortunately, we were not able to find any good way to discover a new hidden service or to find where a known hidden service is hosted. For more discussion, see sections 7.2.2, **Locating known hidden services** and 7.3, **Discovering unknown hidden services**.

7 (U//FOUO) Possible attacks

(TS//SI) This portion of the paper will be devoted to several possible attacks related to Tor. Their application is mostly speculative, as we have not implemented any of them except for the MJOLNIR-related attacks, and we have not attempted any of them outside

of the lab. These descriptions are intended to be used as possible avenues for future research and development.

7.1 (TS//SI) Denial-of-service-style attacks with MJOLNIR

(TS//SI) An unmodified Tor client will never include a given server in a circuit more than once. This ensures that no server will have to do more work than any other. However, only the client can check for repeated servers. With MJOLNIR we can build an arbitrary circuit of arbitrary length, violating nearly all restrictions on circuit construction with impunity. This capability allows for many interesting DoS-style attacks as we can force other server to do many times the work we have to. Since Tor normally sends packets of length 586, large circuits can heavily tax the Tor cloud in general, and a targeted server in particular.

(TS//SI) The biggest problem with these attacks is that they take us a large amount of work to set up. For an n-server circuit, we must send at least n cells to set up that circuit. Building a circuit can take a large amount of time, especially if the servers are on different continents, as is the case in the Tor cloud. The number of packets generated is on the order of n^2, and the time to set up a circuit grows similarly. Once the circuit is set up, the amount of traffic grows linearly with the number of servers on the circuit.

(TS//SI) The point of building these circuits is to get target servers to handle much more traffic than our client machine while not disturbing the rest of the Tor cloud. Once the circuit is set up, this is not a problem. For example, if we set up a circuit in which one server appears 50 times, it will always do at least 50 times more work than our machine once the circuit is established. We can use this for two distinct types of attack.

(TS//SI) First, there is the possibility that if we had enough circuits running through a target machine, we could cause a socket denial of service. Each connection between two Tor servers uses distinct sockets. Once two machines are connected to each other, they will always use the same sockets, but with enough machines connecting to a server, it is theoretically possible to cause a machine to use all of its available sockets.

(TS//SI) Also, Tor gives servers the option of sending only a limited amount of traffic through the Tor network in a given period (day, week, or month). By targeting a machine with these attacks, we could cause a server to hibernate, essentially taking it out of the Tor cloud until the next accounting period begins. This could also make bandwidth costs prohibitively high for targets.

7.1.1 (TS//SI) Coil attack

(TS//SI) Our first implemented technique for achieving either type of attack is what we call the “coil” attack. For this attack, we choose two target servers and set up a circuit that bounces back and forth between the two, “coiling” them around each other. When we set up a circuit of n_c coils, the circuit will actually be composed of $2n_c$ servers. This is because for each coil both targets are included once. This attack is concentrated, but requires two targets in order to work. This may be detected by an observant sysadmin, but by default,

TOP SECRET//COMINT//20291123

Tor's logs do not store IP addresses, so the information would have to come from elsewhere. Even given the necessary IP information, the system administrator will see only the other target sending and receiving large quantities of information. Without further investigation and some creative thinking, it is highly likely that the system administrator will assume our other target is actually launching the DoS attack. We can leverage the anonymity provided by Tor to protect ourselves from easy detection by simply appending this coil onto the end of a circuit.

7.1.2 (TS//SI) Flower attack

(TS//SI) Our second implemented technique for achieving a DoS attack is what we call the "flower" attack, and it targets only one server. In this attack, MJOLNIR builds a circuit to a random server, then to the target, then another random one, then the target, and so on for a specified number of "petals." When we set up a flower with n_p petals, the circuit will actually be composed of $2n_p$ servers. The flower attack distributes most of the load amongst the machines of the Tor cloud, so at most the target server will suspect that it is being attacked. But even if the target knows about an attack, it will have to assume that someone is trying a distributed denial-of-service attack. While the system administrator will likely recognize that his attacker(s) are leveraging Tor, Tor's strong anonymity will leave him with few options. He will be forced to accept it or reduce/eliminate Tor usage of his machine.

(TS//SI) Perhaps the most serious flaw with the flower attack is that administrators on the petals might notice that their Tor traffic is coming from and going to the same server, a violation of Tor's protocol which clearly requires a custom client. However, the petals still would not have enough information to know where the request originated, and any attempt to discover this would necessarily break the anonymity of Tor. In order to get around this, we could build a petal with two servers between the instances of the target server. In this way, no server realizes that it is taking part in an attack.

7.2 (U//FOUO) Traffic analysis

(S//SI) Is there a way to undo the anonymity that Tor provides? It does not appear that Tor leaks any significant information about the client to anyone other than the client. However, it is still possible to analyze the pattern of traffic in and out of the Tor cloud to try to determine which clients are responsible for which requests. Tor does not make such an attack (here referred to as a correlation attack) easy.

7.2.1 (U) A basic Tor circuit

(U) The most common activity in Tor is carried out across circuits 3 nodes long. A client connects to a first server, which connects to a second server, which connects to a third server, which connects to the desired web page or other service.

(TS//SI) For this scenario, we are most interested in tracing a request back to the client, assuming that we see an exit node connect to something we deem interesting. The worst (and most likely) case is that we will not own any of the servers along the way. There are two types of potential correlation attacks, although both are complex.

(S//SI) First method: circuit tracing This method requires data on all of the connections for each server in the attacked circuit. While currently unrealistic in general, this is a potentially powerful attack. A small, private Tor network, for example, could possibly be completely compromised by this technique.

(U//FOUO) We also need to have a good idea of how long it takes to send data across the network to another machine. This comes with the usual caveats that network congestion, distance between links, and packet size affect transmission time. Although in theory Tor packets are all the same size owing to the fixed cell size, in practice we see different packet sizes, as two or more cells sometimes get sent in the same packet. Despite this, we should have a good idea of the network latency. Actual processing time devoted to the cells is minimal, and can be ignored.

(S//SI) We begin by focusing on an exit node making a connection to material of interest. At this point, we must keep track of all of the connections that node has up. When the exit node receives data, it will send it back down the circuit that requested it. Thus data comes in to the exit node, and then the node will send out data over the appropriate connection.

(S//SI) Unfortunately, the exit node will almost certainly be sending data out over several connections. We must keep track of all of them. Let the network latency be x seconds. When we see data get sent to the exit node, we only need to pay attention to the outgoing connections that send out data roughly x seconds later. We must take into account some error in our timing, to allow for differences in latency owing to differences in network conditions. We then make a list of all of the nodes in the Tor cloud that the exit server sends data to. These are all potential second nodes.

(S//SI) The above process must be recursively performed on the list of potential second nodes. Hopefully this number of nodes will not be prohibitive, but we have no hard data about how many nodes a given machine in the Tor cloud is simultaneously connected and actively sending data to. Again, we will use the timing to determine which connections are of interest. This is made slightly easier by the fact that no node will appear twice in a circuit, meaning that we can ignore any outgoing connections from our possible second nodes to our exit server. We can also ignore any connections made to machines outside of the Tor cloud (i.e. clients) because the second server should only be talking to other Tor servers. Now we have an even larger list of nodes to look at, and these are our possible entry servers.

(S//SI) Finally, we look at the outgoing connections from the entry servers that are sending out information roughly x seconds after the possible second nodes. This is made easier by the fact that we are really only interested in connections into the Tor cloud from outside of the Tor cloud, so we can ignore any connections made to other Tor servers. So

now, we have a large list of potential circuits.

(S//SI) From here, we watch for more interesting material to be sent to the exit server and repeat the process. We should get a different set of potential circuits, as not every connection should be active again at roughly the same time except the connections along the circuit for which we are looking. There should be some overlap between the two lists. This intersection is our new, smaller list of potential circuits. After repeating this process enough times, we should only have one circuit, or at most, a handful.

(TS//SI) This analysis becomes easier for each box that we own along the circuit, since we then know for sure which incoming connection matches which outgoing connection on that box. This means that our tree of potential circuits branches one less time. Owning the exit server is probably most useful for this type of analysis, since it is the only machine known to be a part of the circuit from the beginning.

(TS//SI) There are some difficulties. How many potential circuits will result from each iteration? Without some data on the real Tor cloud regarding how many connections each server is sending data over for a given span of time, it is impossible to say. It would not be surprising if this number turns out to be very, very large, even with the pruning at each step. Assuming the number of potential circuits is not too large to work with, how many iterations of this algorithm will it be necessary to run before we can track down the one, true circuit? Again, without some real data about the Tor cloud it is impossible to say. Clearly, the more potential circuits we find at each iteration, the more iterations it will take to narrow the list down to one.

(U//FOUO) The biggest problem with this idea is that Tor will only use a circuit for a short period of time before tearing down and building a new one. Currently, the default period of time for this is 10 minutes, and Tor checks every 30 seconds to see if any circuits need torn down. So at best there are 10 minutes during which one can perform this analysis.

(TS//SI) Second method: Black-box the Tor cloud In order for this attack to work, we only need to see connections entering and exiting the Tor cloud. When we see an exit server make a request of a service that we are interested in, we keep track of when information is sent back into the Tor cloud. We then keep track of each client connected to the Tor cloud that receives data afterward for an arbitrary amount of time, probably a TCP timeout. This will likely be an enormous number of clients. Each client will have a certain probability of being the one that made the request; for example, a client that receives information 2 seconds after the service sends back some data is more likely to be the one that requested it than a client that receives information 10 seconds after the service sends the data. Each time we see the service send information back to that same exit server, we track which clients receive data during the right time frame. Each iteration will give us a large number of clients, but we are only interested in the clients that receive information each time, that is, clients in the intersection of all the sets. The client making the requests must be in that intersection. The client with the highest total probability over all the iterations is the most likely to be the target.

(TS//SI) This technique is essentially a wide-scale version of the circuit-tracing technique, and is less precise. However, it should be easier to collect this information, as there are currently only around 450 publicly known exit servers, and it is easier to watch connections from 450 machines to the outside of the Tor cloud than it is to watch all connections of all machines, as the circuit-tracing method requires. The hope is that the large amount of data we could collect for this method would compensate for its less precise nature.

7.2.2 (S//SI) Locating known hidden services

(S//SI) In the case of a hidden service, the most likely scenario is that we know about a hidden service and to want to know where (i.e. on what machine) it is located. We would use a Tor client and connect to the hidden service in order to set up a circuit. See appendix C for a description of hidden service circuits. For a normal hidden service circuit, circuit tracing would have to stretch across a circuit containing five servers, making it too complex, and we can not use the Tor black box approach. Things must be done somewhat differently.

(TS//SI) If we know the rendezvous point (RP), the analysis becomes more manageable. We can set a preferred RP using the configuration file of even a basic Tor client. This will work for our purposes, but if we do that, then we will be using a normal Tor server as a RP, meaning that it will have several connections through it at a given time. If we program a custom client to choose an arbitrary RP, perhaps even making the client box the RP (see section 8.1), then we can choose a RP outside of the Tor cloud, meaning that the only connections through it will be ones related to the hidden service. For this attack, assume that we know the RP, as well as the server that it is connected to for this circuit.

(S//SI) When the client makes a request, it will go to the RP, which will send it down another circuit to the hidden service. This gives us a chance to to see what node the RP is connected to (call it the first node). Then, we check the timing in the same manner as in the basic Tor circuit to see to which servers it is connected. We can ignore any connections that go outside of the Tor cloud. We now have a list of potential second nodes. In a similar fashion, we check the timing of all of these to see to which servers they are connected. One of these is the hidden service.

(S//SI) The hidden service will respond, so we will see data come back up the circuit to the RP. This gives us another chance to work with the timing to try and figure out the circuit. It would be easiest to figure out the second node during this step. One would have to see which machines are talking to the first node just before it talks to our RP, and correlate that list with our possible second nodes from when the request was sent down the circuit. The actual second node should be in the intersection of these two lists.

(S//SI) As with the normal Tor circuit, this will likely give us many potential circuits, and need to be iterated several times before the hidden service is located. There are two main advantages over the same analysis for a normal Tor circuit. The first is that there are fewer servers between the RP and the hidden service, meaning that we only need to keep track of two possible branches in the circuit, instead of three. The second is that we

can ask the hidden service to connect to the RP as many times as we want, whenever we want. This allows us to perform the correlation over and over again, albeit with different circuits each time. The more data we have, the more likely we are to locate the hidden service.

(S//SI) We know that the hidden service will never build a circuit to the RP that contains the hidden service machine, although it can be its own RP. Thus, whatever machine the hidden service is on will never be the machine that connects to the RP. Every time we ask the hidden service to build a circuit to the RP, we keep track of which machines are connecting to the RP. Eventually there will be a sizable list of machines that cannot be the hidden service. It is doubtful that we will narrow it all the way down to the machine on which the hidden service is located, but it will help us prune the possible branches from possible second nodes.

(S//SI) The hidden service may have a list of nodes that it will never use in a circuit (under *ExcludeNodes* in the configuration file); these will never show up as the machine that connects to the RP. It will, however, use a server listed under *ExcludeNodes* as an RP. Also, the hidden service may have set up one or more entry guards for its circuits, meaning that the first node from the hidden service (the second node in the above example) will always be one of those routers. If it has only set up one entry guard, that machine will also never be the one that connects to the rendezvous point. This is unlikely, because most of the time one sets up multiple entry guards. A fixed first node effectively shortens your circuit by one hop. So this analysis may turn up the hidden service's excluded nodes and entry guards even if it does not locate the hidden service.

7.3 (S//SI) Discovering unknown hidden services

(S//SI) We did not discover any method to detect previously unknown hidden services. If a hidden service is being offered on a machine that is also a Tor server, there may be some noticeable difference between its traffic and that of a typical Tor server. However, a hidden service may be offered on a machine outside of the Tor cloud.

(S//SI) In our time in the lab, we found that running an nmap on a node that is offering a hidden service will turn up the port that the hidden service is using to deal with incoming connections. It can then be directly connected to, outside of Tor.

(TS//SI) There are several issues with this. We must already suspect the machine of housing a hidden service to decide to do the nmap, or alternately, nmap several servers in the hopes of turning up a hidden service. Then, the machine running the hidden service will likely have several ports open, and there is no way to tell which port is offering a hidden service just by looking at the list of open ports. A hidden service may even be offered on a well-known port such as port 80. We would have to try to connect to each of the ports we see open on a machine to determine if there is a hidden service being run. We would not even know which protocol the hidden service is running. It may be an HTTP server, an FTP server, an SMTP server, etc. The only thing we know is that the protocol must run over TCP. It is not enough to attempt to connect once to each port, using an HTTP GET request. Several protocols must be tried.

(S//SI) Also, keep in mind that the people running these hidden services are generally very careful, and all of these requests and nmaps are detectable by a canny system administrator. We can hide ourselves by using Tor to route our requests, even if we don't try to access the hidden service the Tor way, but the attempt may well be detected. This, combined with the shot-in-the-dark nature of the whole enterprise, limits its usefulness.

7.4 (U//FOUO) Man-in-the-middle attack

(TS//SI) Because Tor's X.509 certificates are self-signed, it may be possible to perform a man-in-the-middle attack using MJOLNIR to forge certificates. Assume that we wish to intercept traffic between two Tor servers with the nicknames Alice and Bob. They will send X.509 certificates that are structured in a very specific way; see section 5. We can use MJOLNIR to create our own Tor-style X.509 certificate with "Alice" or "Bob" in the correct location and appear to be the other machine. From here, we carry out a man-in-the-middle attack in the normal fashion.

(TS//SI) However, this only allows us to get under the layer of TLS encryption. As covered in section B.3.2, the client sets up a layer of encryption with the server (for example, Bob) by passing it an "onion skin". Most of the time, this is done using a CREATE cell which contains half of a Diffie-Hellman handshake. The contents of this CREATE cell are encrypted to Bob's publicly advertised RSA key. This means that we won't be able to read the CREATE cell, and we still won't be able to read the contents of the cells that go through the man-in-the-middle. All we will be able to see is 3 bytes of header on the cells that we would relay through.

(TS//SI) If the man-in-the-middle intercepts a CREATE_FAST cell, it will be able to read the X value that it contains and create a shared secret with the client, in effect becoming the first server in the circuit. The intended first server can be completely removed from the circuit, and the man-in-the-middle will see which server is second in the client's circuit.

(TS//SI) We did not test this in our work, so there may be issues with this attack that we did not foresee. At the very least, there is still the obstacle of the authoritative directory servers. They act in a fashion similar to a trusted third party. The directory listing weakly associates specific RSA public keys with specific server nicknames. If a server operator registers their Tor server with the authoritative directory server (this must be done out-of-band), then this becomes a strong association, and certificates with Alice's name will not be accepted unless they have Alice's listed public key. Thus, it may not always be possible to perform this attack. However, the authoritative directory servers appear vulnerable. See section 8.4 for further discussion.

8 (U) Future areas of study

(TS//SI) There are still many interesting and important areas of Tor functionality to explore. As with any software project, there are always bugs to fix and features to add. There

are also two major avenues of exploration in the architecture of Tor itself that warrant further study because they may provide ways to discover important target information.

8.1 (U//FOUO) Expanding MJOLNIR's functionality

(TS//SI) The Tor code provided a solid basis for our work on MJOLNIR, but every modification opens up the opportunity for bugs to arise. All of our work was done in a small lab with eight machines, so the MJOLNIR code must be tested on a larger network to see if it still behaves correctly, or if its response time is simply too long to be useful. There remain large portions of code that are only useful to servers that can safely be cut from the MJOLNIR main code. Unfortunately, we did not have enough time to completely divorce the client and server sides of Tor's code.

(TS//SI) MJOLNIR still has one major deficiency: it cannot connect to hidden services. In order to successfully connect to one, the client must set up a circuit to a rendezvous point, then to an introduction point, then must request that the hidden service meet it at the rendezvous point. Doing this much is very important, and with the right modifications, it becomes possible to have a client be its *own* rendezvous point. As such, there will be only two servers between our client and the hidden service, making a traffic analysis attack easier, which may open the way for locating the identity and location of hidden services.

8.2 (U) Privoxy

(U//FOUO) Tor is useful only for rerouting traffic so that one's location and/or browsing habits are kept secret. Tor does not examine the packets it sends for identifying information such as IP addresses. Because of this, Tor is packaged with Privoxy, a free software filtering proxy. Privoxy's purpose is to scrub all identifying information from HTTP requests, etc. Any Tor user who is not using a filtering proxy like Privoxy is probably leaking identifying information, especially in DNS requests. There may be situations in which Privoxy does not hide all of the information that needs to be hidden for perfect anonymity. However, we did not have time to study the interaction between Tor and Privoxy, and it warrants some attention.

8.3 (U) Tor remote control

(TS//SI) One area of interest that we looked at briefly before turning to other, more useful, ideas was the possibility of using Tor's control port to control a client or server. Tor offers the option to set up a control port. A control port would allow one to remotely control any standard configuration option of his Tor server as if he were directly editing Tor's configuration file. However, this did not offer the flexibility or power that we needed to set up a custom client, so we abandoned the idea early on. This does not mean that there is no useful application of using the control port. For example, one may be able to set up a carefully selected set of entry guards or exit servers (or both) so that suspected target

traffic will always route through friendly servers. There may be other ways of exploiting the control port, including going so far as to modify the target's list of trusted directory servers. This might send the target client to a directory server in which all of the servers are friendly.

8.4 (TS//SI) Directory server exploitation

(TS//SI) It may also be useful to study Tor directory servers in more detail. Our work focused solely on the client, but many attacks would be much easier with access to more Tor servers. The directory servers ultimately control which Tor servers are used by clients. We have found that a server can put itself on a directory server multiple times; all it takes is the server running several Tor processes, each having a different nickname, open port, fingerprint, and log file. This only requires different configuration files for the different processes, which are easy to set up. That machine will handle a disproportionate amount of traffic, since it is listed several times. This increases the density of friendly servers in the cloud without increasing the number of servers we have set up. Unfortunately, each listing has the same IP address, which would be very noticeable to anyone who inspecting the directories.

(TS//SI) A more useful attack would be to fool Tor clients into believing that a friendly box is actually someone else. This could potentially be accomplished by getting the directory servers to list the friendly box as the other machine. That may allow us to see more information than the target wants us to see. For example, if we know that a certain target has a list of preferred entry guards and exit servers, and if we can list our machines as those servers, then the target (or second node, respectively) will be very likely to send traffic through our machines, making it easier to correlate the traffic and determine what the target is requesting and/or where the target is.

(TS//SI) There may be many other applications of this type of attack, and for this reason, directory server exploitation should be looked into.

9 (U) Acknowledgments

(U//FOUO) We would like to thank the CES Summer Program for giving us the opportunity to work on this problem. We would also like to the our mentor, for his guidance and help with this problem. Also thanks to the CES Summer Program directors, and the other students in the program. Their questions and ideas provided us with several ideas for areas of study.

10 (U) References

1. Dingledine, Roger. "TC: A Tor control protocol." <http://tor.eff.org/cvs/tor/doc/control-spec.txt>. Accessed 2006-03-10.

2. Dingledine, Roger. "Tor directory protocol for 0.1.1.x series." <http://tor.eff.org/cvs/tor/doc/dir-spec.txt>. Accessed 2006-03-10.

3. Dingledine, Roger. "Tor Rendezvous Specification." <http://tor.eff.org/cvs/tor/doc/rend-spec.txt>. Accessed 2006-03-10.

4. R. Dingledine and N. Mathewson. "Tor: manpage." <http://tor.eff.org/tor-manual-cvs.html.en>. Accessed 2006-03-10.

5. R. Dingledine, N. Mathewson, and P. Syverson. "Tor: The Second-Generation Onion Router." <http://tor.eff.org/cvs/tor/doc/design-paper/tor-design.html>. Accessed 2006-03-10.

6. "TheOnionRouter/TorFAQ." <http://wiki.noreply.org/noreply/TheOnionRouter/TorFAQ>. Last edited 2006-03-10.

7. "Tor Linux/BSD/Unix Install Instructions." <http://tor.eff.org/doc/tor-doc-unix.html>. Accessed 2006-03-10.

8. "Tor Protocol Specification." <http://tor.eff.org/cvs/tor/doc/tor-spec.txt>. Accessed 2006-03-10.

9. "Tor Server Configuration Instructions." <http://tor.eff.org/doc/tor-doc-server.html>. Accessed 2006-03-10.

A (U//FOUO) Tor glossary

Cell: Tor's basic building block of communication. Any message either sent to or from a client or server will communicate in cells, each of which is a fixed length (currently 512 bytes); any message longer than this is broken up into multiple cells.

Circuit: a series of Tor servers that transfer data for a client to a destination. Each server in the circuit has its own level of encryption so that no server can read the message except the last one.

Directory: the collection of information about servers that are available for use as Tor nodes.

Directory server: a server that keeps the directory information. These can be either trusted (hard-coded into the client or specified in the configuration file), or mirrored on normal Tor servers.

Entry Guard: a client or server can specify servers that it wants as its first node in its circuits. These are entry guards.

Exit Server: the last server in a circuit. Clients can specify preferred exit servers.

Hidden Service: A service hosted anonymously using Tor. The IP of said service need not be revealed to anyone accessing it. For more details, see appendix C.

Server/Router/Hop/Node: a computer running Tor that is connected to the Tor cloud. Servers have nicknames and should be known from the directory.

Tor Cloud: the set of servers running Tor. When a client or server builds a circuit, it is connecting to the Tor cloud.

B (U) Programming details

B.1 (U) Building circuits

(U) The most important structure in Tor is the circuit, that is, the list of machines that a client will send its traffic through. Information regarding this circuit is maintained in the **circuit_t** structure. This structure stores information regarding the connections and encryption used by the machine when sending information along the circuit. For the client, it also contains a list of the servers along the path and the encryption information to use with each server.

(U) When a user wants to build a circuit, most of the work happens in the function `circuit_establish_circuit`. It is passed five parameters: *purpose*, *need_capacity*, *need_uptime*, *internal*, and a pointer to *info*. These parameters contain information used to determine several aspects of the circuit that is ultimately built. In particular, *info* contains

information for connecting to a specific exit server if one has already been chosen. (This only happens when attempting to access one of **Tor's hidden services**; see appendix C below.) `circuit_establish_circuit` is a small function, consisting of only 20 lines. All of the real work happens in 4 function calls that it makes.

(U) First, `circuit_establish_circuit` calls `circuit_init`, to which it passes *purpose*, *need_uptime*, *need_capacity*, and *internal*. Right away, this function calls another function, `circuit_new`. This function allocates memory for the **circuit_t** structure, initializes several fields of the structure, calls `circuit_add` to add the new circuit to the global circuit list, and returns. Back in `circuit_init`, the parameters are used to initialize more fields in the **circuit_t** structure. Next, it sets the state of the circuit to `CIRCUIT_STATE_OR_WAIT`, indicating that it is waiting to establish a connection to the next server in the path before it sends along any packets. Finally, `circuit_init` returns a pointer to the newly-created `circuit_t` structure.

(U) Back in `circuit_establish_circuit`, the function `onion_pick_cpath_exit` is called and passed the pointer to the new **circuit_t** from `circuit_init` and the parameter *info* originally passed to `circuit_establish_circuit`. `onion_pick_cpath_exit` starts by getting the list of available routers using the function `router_get_routerlist`. This router list is created using information obtained from the directory servers. The process is not covered in this paper; see section 8.4.

(U) Next, the function `new_route_len` is called to determine the number of servers that should be in the circuit. This number is based on the purpose of the circuit and whether or not *info* is NULL, but for most Tor circuits, it will be three. After a number has been decided on, the function `count_acceptable_routers` checks how many servers it thinks can route traffic. If the number of acceptable routers is less than two, the function returns failure, and no circuit is built. If the number of available routers is two or more, but less then what was decided on, this becomes the length of the circuit.

(U) One of two things can happen next. If *info* is not NULL, then the server with that information will be used as an exit, and the program moves on. Otherwise, the program picks a good exit server. This is done in the function `choose_good_exit_server`, which takes parameters *purpose*, *dir* (a router list), *need_uptime*, *need_capacity*, and *is_internal*. Three of these are parameters originally sent to `circuit_establish_circuit`. There are two cases, based on the circuit's purpose. For general purpose circuits, if *is_internal* is set (meaning the circuit is intended to be used to connect to a server within the Tor cloud, most likely as a rendezvous point for a hidden service), then the exit server is chosen using `router_choose_random_node` (see **Choosing a hop**, section B.1.1). Otherwise, `choose_good_exit_server_general` chooses the exit server. This function takes a list of all available exit servers, discards any that are currently unsuitable (for example, those servers that are listed as down), and selects from those remaining one that can handle the most pending connections. It then returns the router information necessary to connect to the selected server.

(U) After initializing the circuit and picking out an exit, the other hops in the circuit must be chosen. To do this, the program calls `onion_populate_cpath`. This function repeatedly calls another function, `onion_extend_cpath`, until the whole circuit is built. In

`onion_extend_cpath`, `choose_good_entry_server` is called to select the first node in the circuit. This takes the entire list of available servers, gets rid of those unsuitable (already chosen as the exit, listening on wrong port, etc.), and from the remainder, chooses a random node using `router_choose_random_node`.

(U) For the middle hop, `choose_good_middle_server` is called. Essentially, this is the same routine as `choose_good_entry_server` except that it also excludes from the list of possible routers those that have already been chosen as other hops in the circuit.

(U) For each hop along the circuit, once a good router is chosen, the choice is returned to the calling function. The router is then appended onto the circuit by calling `onion_append_hop`. This function first calls `onion_append_to_cpath` which adds this router to the list of routers in the circuit. How does the program keep track of this list? One of the elements of the **circuit_t** structure is *cpath*, which is actually a pointer to a **crypt_path_t** structure. All of the information necessary to connect to each router, including the IP address and cryptovariables between the two machines, can be found in this structure, as well as a few simple bandwidth accounting fields. *cpath* points to the first router in the circuit, but it also maintains *next* and *prev* members in the **crypt_path_t** structure, which, unsurprisingly, point to the next and previous routers in the circuit, respectively. It is important to note that not only is this a doubly linked list, but it is also circular.

(U) After all three routers in the circuit have been chosen and appended to the circuit, the program returns all the way back to `circuit_establish_circuit` and moves on to the next important step, establishing connections to all of the routers.

B.1.1 (U) Choosing a server

(U) Most of the time, `router_choose_random_node` is called to choose which server will be used. Despite its name, the choice is not exactly random, and the function is passed several parameters to help with its decision. The most important arguments would be the list of preferred servers, and the lists of excluded servers. For some reason, two lists of excluded servers are passed: a character string containing server nicknames, and a **smartlist_t** structure see below containing server information. The lists do not necessarily overlap. Several flags indicating user preferences regarding server capacity, uptime, and the like are also passed. One of the arguments, *strict*, is set if the user will only accept a choice from their list of preferred servers.

(U) The function begins by creating a smartlist of the preferred servers, from which it removes any excluded servers. It will then call `smartlist_choose` to select a random element of that list.

(TS//SI) `smartlist_choose` is a very simple function. In Tor, the **smartlist_t** structure is essentially a resizable array, and several functions are provided to manage them. (See appendix E, **MJOLNIR API** for more on smartlists.) `smartlist_choose` generates a random index and returns the element of the smartlist with that index. The random index is generated using the OpenSSL `RAND_bytes` function. Care is taken to avoid bias in the random numbers, by making sure that the maximum value returned by

`RAND_bytes` is a multiple of the size of the smartlist. For example, suppose the size of the array is 10 and `RAND_bytes` could return any value up to 25. Then when computing a random index for the smartlist, the function would generate a value less than 25 and take it mod 10. Because 25 is not a multiple of 10, the numbers 0, 1, 2, 3, and 4 would occur more often than 5, 6, 7, 8, and 9.

(U) If for some reason no preferred server is chosen (for instance, no preferred servers were specified), two things can happen. If *strict* is set, the function returns NULL, indicating failure. Otherwise, the program continues by attempting to find servers that meet the preferences indicated by the flags. This is done by creating a smartlist containing all running routers that meet the preferences, which is passed to `smartlist_choose`. The only exception is if the parameter *need_capacity* is set, in which case `routerlist_sl_choose_by_bandwidth` is called, which makes its random choice slightly differently.

(U) `routerlist_sl_choose_by_bandwidth` works differently than `smartlist_choose`. For the given list of servers, it adds up all of the bandwidths. At the same time, it creates a separate smartlist of each server's bandwidth. For example if the first three servers can handle 200 bytes/s, 450 bytes/s, and 1000 bytes/s respectively, then the smartlist will have entries 200, 450, and 1000. Then, a random number between 0 and the total bandwidth is selected (using the same OpenSSL function as in `smartlist_choose`). It steps through the smartlist of bandwidths, summing the bandwidths, and chooses the first server where the bandwidth sum is greater than the random number. Again returning to the example, if the random number was 412, the second server would be chosen, because $200 + 450 = 650$, and $650 > 412$. If the number was 70, the first server would be chosen. We see that higher bandwidth servers are more likely to be chosen than lower bandwidth servers, as one would expect.

(U) If the function is still unsuccessful in finding an acceptable server, the program tries one more time to find a next hop, this time without setting any preferences. This is the last resort; if no server can be found, then the function returns NULL and the circuit fails.

B.2 (U) Connecting the circuit

(U) Recall that a client records all of the servers in its circuit using the **crypt_path_t** structure. Once it has decided upon a complete circuit, the client must now connect the servers in the circuit so they can send information back and forth. In Tor, the **connection_t** structure holds the information needed to set up such a connection, such as the IP address of the node it connects to, the port it should connect on, and a lot of other information. A connection is specified by an id number. Connections and circuits are related in complex ways. A single connection may be part of several different circuits, and each **circuit_t** structure holds information on a previous and next connection (*p_conn* and *n_conn*, respectively).

(U) After all the servers that will make up the circuit have been determined, `circuit_establish_circuit` calls `circuit_handle_first_hop`. First, this function re-

trieves information regarding the first server from the circuit's *cpath* variable. This information is used to see if there is already a connection to that server. If so, that connection will be used for the circuit's *n_conn*. The appropriate fields of the **circuit_t** structure are set to hold information about that connection, and `circuit_send_next_onion_skin` is called to set up the cryptovariables between the two routers. See B.3, **Encryption** for more about the information sent in an "onion skin".

(U) If there is no pre-existing connection to the first server, the function `connection_or_connect` is called. It takes as arguments the address, port, and identity digest of the server being connected to. The identity digest is a SHA-1 hash of the server's public RSA key, and is often used to uniquely identify a particular router. A lot of things happen in this function, with the ultimate goal of returning a fully initialized connection. First, it calls `connection_new`, which allocates memory for a new connection and initializes various members of the **connection_t** structure.

(U) Next, `connection_or_connect` calls `connection_or_init_conn_from_address`, which is passed the newly created **connection_t** structure, and the address, port, and identity digest of the server being connected to. It tries to retrieve the server's information based on its identity digest, which, if found, is used to fill in more variables in *n_conn*. If it could not find the server information, then the function will fill in the necessary information itself, occasionally using more generic information than would have been retrieved based on the identity digest. With all of these things initialized, the function will return to `connection_or_connect`.

(U) The next important step is that `connection_connect` is called. This attempts to create a non-blocking socket connected to the IP address:port combination it was passed. Most of what goes on in here is basic C socket calls. If successful, the connection is added to the global connection array, and read and write events for the connection are set up, using libevent. The events on the connection are initially set to be non-reading and non-writing, but as the program continues, the events will be changed to handle reads and writes.

(U) Finally, `connection_or_connect` will call `connection_or_finished_connecting`, which will start the TLS handshake between the two servers. After the TLS handshake completes, the client will send a CREATE cell to set up encryption between itself and the first server. All of this will be discussed in the next section, **Encryption**. So once `connection_or_finished_connecting` returns, everything up to and including `circuit_establish_circuit` returns.

(U) At this point, the circuit is still not fully connected. The client has set up its connection to the first server, and is waiting for a CREATED cell in response to the CREATE cell it just sent. Once a CREATED cell is received, the client and the first server have finished agreeing on their cryptovariables, and can communicate freely under a layer of encryption. At this point, the client will make an EXTEND cell, which contains all of the information necessary to connect to the second server in the circuit as well as cryptovariable information for the layer of encryption between the client and the second server, and send it encrypted to the first server.

(U) The first server will decrypt the cell, and read the information for connect-

ing to the second server. If there is already an existing connection between the two servers, that connection will be used for communication along this circuit. If not, then `connection_or_connect` is called to open a connection between the two; see above. Once a connection is open, the first server packages the cryptovariable information for the second hop in a CREATE cell and sends it to the second server. Upon receiving a CREATED cell from the second hop, the first server packages the necessary information into an EXTENDED cell, which it sends back to the client.

(U) This whole process is repeated one more time, as the client sends an encrypted EXTEND cell through the first server to the second server, which will send a corresponding CREATE cell to the third server. Again, an EXTENDED cell gets sent back to the client after the third server sends back a CREATED cell. Once the client has finished setting up the cryptovariables between itself and the third server, all of the connections in the circuit have been made, and it is considered open. Any applications waiting to send packets through Tor are notified that there is a new circuit available to use.

B.3 (U) Encryption

(U) Tor packets are sent under several layers of encryption. Each connection between two machines is under TLS encryption. Additionally, each Tor payload is encrypted by the client once for each hop in the circuit, using a symmetric cipher.

(U) In Tor, all information to set up the symmetric cipher between the client and the servers in the circuit is sent in what is known as an "onion skin". In a basic Tor circuit, three onion skins are sent from the client, one for each server in the circuit. Each server reads its onion skin, sets up its cryptovariables, and sends back information to the client to allow the two machines to share cryptovariables. In this way, the client gets the necessary information to put its data under one layer of encryption for each server in the circuit. All cells across a typical Tor circuit will be under three layers of encryption when it is sent, and each server will strip off one layer. See figure B.3 for a visual representation of this process. It is these layers of encryption that give Tor its designation as an "Onion Routing Network". The last server in the circuit will have the cell in the clear and be able to pass along the request found inside.

B.3.1 (U)TLS encryption

(U) Tor's TLS handshake is carried out in 3 main phases. The function `connection_or_finished_connecting` (see above, **Connecting the circuit**) calls `connection_tls_start_handshake`, which allocates memory for a new Tor TLS context and initializes several of its fields, including the socket that will be written to and the SSL_CTX that will be used for communication. The SSL_CTX is initialized at Tor startup, and is changed every 2 hours. It is this structure that contains the certificates that are used during the TLS handshake, as well as the list of cipher suites to be used. Tor's certificates are discussed in more detail in section 5, **Tor's X.509 certificates**. By default, Tor supports 2 cipher suites:

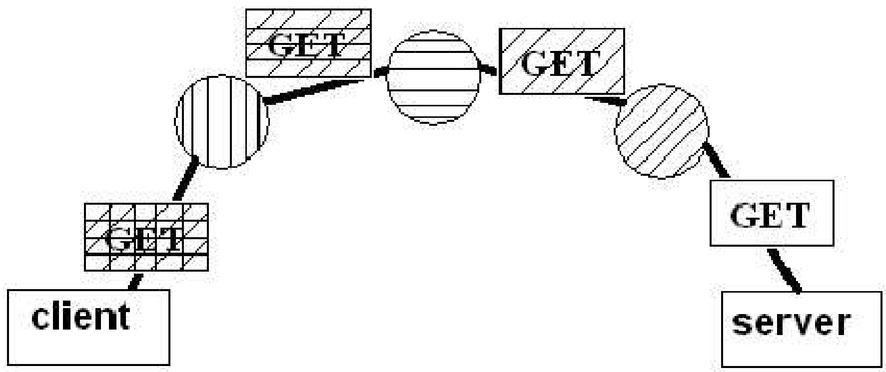


Figure 2: **(U//FOUO) Layers of encryption on a normal GET.**

TLS_EDH_RSA_WITH_DES_192_CBC3_SHA

TLS_DHE_RSA_WITH_AES_128_CBC_SHA (this is preferred)

(U) Once the TLS context is properly initialized, the TLS handshake continues in `connection_tls_continue_handshake`, which calls `tor_tls_handshake`. This calls `SSL_connect` or `SSL_accept`, depending on which side of the handshake this is on. Assuming there are no errors, the program moves on to `connection_tls_finish_handshake`, which validates the handshake and extracts the identity key from the certificate. If everything checks out, then the server is considered authenticated and future communications will take place under TLS encryption.

B.3.2 (U) Onion skins

(U) After the client picks out its circuit and sets up a connection to the first server in the circuit, it will send an onion skin to the first server. This is held in the payload of either a CREATE or a CREATE_FAST cell. What's the difference? When the client connects to the first server, it does so using TLS. This means that it has already authenticated the first server's identity and negotiated a key. This allows it to send a CREATE_FAST cell, which does not require an RSA operation nor a Diffie-Hellman operation, since most of the necessary information has already been established. Note that a CREATE_FAST cell can only be sent between a client and the first server in the path, because all other servers need to be authenticated, and the client has no TLS connection directly to them. For this reason, CREATE cells are sent for all other servers in a circuit.

(U) CREATE_FAST cells are simple. The cell contains only a random X value, created using Tor's `crypto_rand` function, which calls the OpenSSL function `RAND_bytes`. The client sends this to the first server in its circuit. When a server receives a CREATE_FAST cell, it extracts the X value and calculates a random Y value using `crypto_rand`.

It sends the Y value back in a CREATED_FAST cell, along with a SHA-1 hash based on $X|Y$, where | denotes concatenation. $X|Y$ is the shared secret and will be used to generate the keys between the two machines. When the client receives the CREATED_FAST cell, it will perform the same hash based on $X|Y$ that the first server should have and compare it to the value in the cell. If the two values agree, then the keys for communication between the two machines are set up. (see **Keys**, below.)

(U) An onion skin in a CREATE cell is more complex. The first 42 bytes are OAEP padding, the next 16 bytes are a symmetric key. This is done using OpenSSL calls. The last 128 bytes of payload are the g^x value for a Diffie-Hellman handshake. The first 128 bytes of this cell (which goes up to about halfway through the g^x value) are encrypted using the second server's public RSA key. The rest of the cell is encrypted with AES in counter mode, using the symmetric key from the first half of the cell.

(U) When a server receives a CREATE cell, it extracts g^x, generates a random g^y, computes the shared secret g^{xy}, and sends back a CREATED cell containing g^y as well as a SHA-1 hash based on g^{xy}. If a machine other than the client gets a CREATED cell (e.g. the first hop receives a CREATED cell from the second hop), it puts the information in an EXTENDED cell, and sends it to the client. When the client gets a CREATED cell or an EXTENDED cell, it calculates g^{xy} and checks that the accompanying hash is correct. The client uses g^{xy} to set up the keys that will be used when it communicates with that node. Note that this key exchange required expensive RSA and Diffie-Hellman operations, in contrast to the simple random number generation of the CREATE_FAST exchange.

(U) Keys (U) When a CREATED or CREATED_FAST cell is received, the function `circuit_finish_handshake` is called. It will extract the shared secret and use it to create keys for communication. Recall that for a CREATED cell the shared secret is g^{xy}, and for a CREATED_FAST cell the shared secret is $X|Y$. Denote the shared secret as K_0.

(U) Regardless of which type of cell is received, the function `crypto_expand_key_material` is passed the shared secret. It does a series of hashes to get $K = H(K_0|[00])|H(K_0|[01])|H(K_0|[02])|$ ..., where | denotes concatenation. This should not include more than 5 hashes (each hash is 20 bytes long). Any extra bytes are ignored.

(U) The first 20 bytes of K form K_H, the handshake digest. This is used to make sure the other node's handshake checks out. The next 20 bytes become D_f, the forward digest, and the next 20 bytes become D_b, the backward digest. These digests are used for integrity-checking purposes. They seed a hash that is used by machines to to determine whether or not a cell has been decrypted all the way or not. D_f is used for client to server streams, and D_b is used for server to client streams. The next 16 bytes of K become K_f, and the next 16 bytes of K become K_b. K_f and K_b are the keys used to encrypt and decrypt the streams of data between the two machines. K_f is used for client to server streams, and K_b is used for server to client streams. All other bytes are discarded.

C (U) Hidden services

C.1 (U) Offering a hidden service

(U) In order to offer a hidden service, the operator of the service (whom we will call Bob) must first add lines to their Tor configuration file indicating the directory containing information regarding the hidden service and the port on which the hidden service will be advertised. After Tor starts up with this new configuration file, it will automatically generate an RSA public/private key pair, and create a `.onion` address for the service. The address is of the form `x.onion`, where `x` is based on the service's public key. Specifically, `x` is the base-32 encoding of the first 80 bits of the SHA-1 hash of the public key.

(U) At this point, Tor picks out at least 3 servers in the cloud to act as *introduction points*. Bob may use the configuration file to specify particular servers that he wants to act as intro points, but if he does not, Tor will simply choose a servers randomly. Once the intro points have been decided upon, Bob does two things. He will generate a service descriptor, which contains information about his public key and how to connect to his introduction points. This service descriptor can be uploaded to the public directory servers, but it need not be if Bob does not wish to have his service publicly known. Bob will also create circuits to his introduction points. Once the circuit is connected, he will send along cells which contain his public key, a hash based on K_H (see **Keys**, in B.3, **Encryption**), and a signature based on that data. These are called RELAY_ESTABLISH_INTRO cells.

(U) When an intended introduction point receives the RELAY_ESTABLISH_INTRO cell, it checks that the information is well-formed and correctly signed. If so, it sends a cell back to Bob acknowledging that it will act as an introduction point for his hidden service. It also makes note that the circuit on which it received the RELAY_ESTABLISH_INTRO cell will be used when dealing with requests for Bob's hidden service, and associates it with Bob's public key.

(U) When Bob receives acknowledgment from his introduction points, he considers the introduction circuits open, he will advertise his service descriptors to the Tor directory servers if he so desires, and he is now ready to receive requests to access the hidden service.

C.2 (U) Accessing a hidden service

(U) There are several ways that a client (whom we will call Alice) can become aware of Bob's hidden service. At the very least, she must receive Bob's public key somehow, most likely through some out-of-band communication such as a phone conversation or encrypted e-mail. Using this, she can request Bob's service descriptor from the Tor directory servers if Bob has chosen to advertise his service publicly. Otherwise, Alice must know Bob's entire service descriptor through some other means.

(U) Once Alice has the descriptor and decides to talk to Bob's hidden service, she must first set up a circuit to some server in the Tor cloud, known as a *rendezvous point* (RP). This point is chosen randomly, although Alice can add a line to her Tor configuration file that will indicate preferred RPs. Once the circuit is established, Alice sends a RE-

LAY_ESTABLISH_RENDEZVOUS cell to the RP, which contains a randomly generated number referred to as a rendezvous cookie (RC).

(U) When the RP receives the RELAY_ESTABLISH_RENDEZVOUS cell, it will send an acknowledgment back to Alice, and make note that the circuit the cell came in on is to be used for a rendezvous, and is associated with that RC.

(U) After Alice receives acknowledgment from the RP, she builds a circuit to one of the introduction points listed in Bob's service descriptor. Once connected, she will send a RELAY_INTRODUCE1 cell to the intro point, which contains a hash of Bob's public key, as well as information intended for Bob that is encrypted to his public key. When the introduction point sees the RELAY_INTRODUCE1 cell, it checks the hash against the hashes of the hidden service public keys that it knows and finds the circuit associated with that public key. It then sends the cell (now encapsulated as a RELAY_INTRODUCE2 cell, with the same payload) down that circuit, to Bob. Afterward, it sends Alice acknowledgment that her cell was sent, and Alice will shut down her circuit to the introduction point.

(U) When Bob receives a RELAY_INTRODUCE2 cell, he decrypts it and pulls out the data Alice sent. This includes information for connecting to the RP, the RC, and Alice's half of a Diffie-Hellman handshake, g^x. If Bob decides to talk to Alice, he will create a circuit to the specified RP. Bob sends the RP a RELAY_RENDEZVOUS1 cell, which contains the RC, Bob's half of the Diffie-Hellman handshake g^y, and the K_H for his interaction with Alice.

(U) When the RP receives a RELAY_RENDEZVOUS1 cell, it finds the circuit associated with the given RC. It then sends g^y and K_H in a RELAY_RENDEZVOUS2 cell down that circuit to Alice. It makes note of the fact that the circuit from Alice and the circuit from Bob are linked, and zeroes out the RC so no one else can attempt to connect to Alice's circuit.

(U) Finally, Alice receives the RELAY_RENDEZVOUS2 cell and uses it to complete the Diffie-Hellman handshake and initialize the cryptovariables between her and Bob. Now Alice and Bob can talk across this newly-formed rendezvous circuit in almost the exact same way as a normal circuit. Neither knows what is on the other side of the RP, so they don't know where the other person's machine is located. The content of their messages is under the normal circuit encryption as well as one more layer based on their shared secret.

D (U) Information at the nodes

(S//SI) It is possible for us to have a friendly computer in the Tor cloud. For example, we can set up a server and advertise it to the Tor directory servers, like all other users. We might compromise another server in the Tor cloud through other means. So how much do we know about the traffic that we are sending and receiving across circuits? Tor does a good job of hiding information about the circuit from all of the nodes in the circuit, in most cases allowing nodes to know only about the machines to which they are directly connected. However, some information is still leaked simply by being a part of the circuit,

and there are some instances where Tor gives the machines slightly more information than is needed. The following section attempts to provide a list of information known by each server (here referred to as a node) along the circuit.

D.1 (U) First node

(U) The first node in the circuit is the only node that knows the location of the machine originating the request. It sets up a **circuit_t** variable with a particular *p_circ_id* and arranges a shared secret with the originating node. (See section B.3.2). When the client extends the circuit to the second node, the first node receives an EXTEND cell with the address, port, and identity fingerprint of the second node. It sets up a connection to the second node and sets *n_circ_id* in its **circuit_t** variable with a new circuit ID. The second node will send back a cell containing its half of a Diffie-Hellman handshake (that is, g^y), and K_H (see **Keys** in section B.3.2 for an explanation of K_H). After this, the first node should not receive any cells that it will be able to recognize, because the cells will be encrypted. In short, the first node knows:

- The location of the client that connected to it.
- Cryptovariables used for communication with the client, which allow it to perform one layer of encryption/decryption on cells to/from the client.
- The ID of the circuit from which the client is sending information. This is *p_circ_id* in the first node's **circuit_t** structure.
- The location of the second node in the circuit.
- The ID of the circuit on which it sends information to the second node. This is *n_circ_id* in the first node's **circuit_t** structure, and will be different from *p_circ_id*.
- g^y in the Diffie-Hellman handshake between the client and the second node, and K_H.

(S//SI) However, a machine may not know that it is the entry node. If the *commonName* on the X.509 certificate passed during the TLS handshake is "client <identity>", then it is definitely an entry node, because it is talking directly to the client. This is sufficient, but not necessary. If the machine receives a CREATE_FAST cell, then it is the entry node, as clients will never send a CREATE_FAST cell farther down the circuit. There is an option in the configuration file that can be set so that a Tor client will never use a CREATE_FAST cell, and Tor servers will never use them, but this still appears to be the most common way that the entry node is set up. Otherwise, a CREATE cell is used. If the node receives a CREATE cell, followed by an EXTEND cell, it may be the entry node. The only guarantee for such a pattern is that it is not the exit server, which will never see an EXTEND cell.

(TS//SI) This knowledge would be useless unless we can induce the client to pick one of our machines as its entry node. In the configuration file, there is an option called

EntryNodes, which is a comma-separated list of server nicknames. Any servers on that list will be preferred when choosing an entry node. If the option *StrictEntryNodes* is true, then only servers from that list can be chosen as entry nodes. If the configuration file can be changed to have a client always use a friendly machine for an entry node, then we can discover who is using Tor, if not what they are requesting.

D.2 (U) Second node

(U) The second node has the least information of any node in the circuit. It does not know who or where the client is, nor does it know what or where the client is connecting to. It does know:

- The location of the machine connected to it.
- Cryptovariables used for communication with the client, which allow it to perform one layer of encryption/decryption on cells to/from the client.
- The ID of the circuit from which the first node is sending information. This is *p_circ_id* in the second node's **circuit_t** structure, and should be equal to *n_circ_id* in the first node's **circuit_t** structure.
- The location of the third node in the circuit.
- The ID of the circuit on which it sends information to the third node. This is *n_circ_id* in the second node's **circuit_t** structure, and will be different from *p_circ_id*.
- g^y in the Diffie-Hellman handshake between the client and the third node, and K_H.

(S//SI) A machine cannot definitely know that it is the second node in a circuit. If it receives a CREATE cell followed by an EXTEND cell on the same circuit, then there is a good chance it is the second node. However, this same pattern is possible for the first node. Because first nodes often see CREATE_FAST cells instead of CREATE cells, it is more likely that if a machine sees such a pattern, it is the second node.

(TS//SI) It is not possible to induce a client to pick our machine as a second node. The second node is chosen using `router_choose_random_node`, with no list of preferred nodes passed to it. The only possible way is to edit the client's configuration file so that their list of excluded servers excludes all boxes except friendly ones, but that seems unrealistic, since that would probably entail listing hundreds of server nicknames in their configuration file.

D.3 (U) Third node

(U) The third node is the exit node for a basic Tor circuit. This means that it sees request that the client makes in the clear, which is useful information. The downside is that the third node has no idea who or where the client is. The exit node knows:

- The location of the machine connected to it.
- The ID of the circuit from which the second node is sending information. This is *p_circ_id* in the third node's **circuit_t** structure, and should be equal to *n_circ_id* in the second node's **circuit_t** structure.
- Cryptovariables shared with the client, which allows it to perform one layer of encryption/decryption on cells to/from the client.
- The exit node can see the data the client is sending, such as an HTTP request or POST data for a form.

(S//SI) A machine can know it is the exit node very easily. There are several functions only called by exit nodes, such as `connection_exit_begin_conn`. There is even a macro, `CONN_IS_EDGE` that will return true if it is passed an exit connection (or an AP connection, which only appears at the client side of the circuit).

(TS//SI) Similar to entry nodes, a client can set up preferred exit nodes in their `torrc` file. There is an option called *ExitNodes* that is a comma-separated list of server nicknames, and any servers on that list will be preferred when choosing an exit node. If the option *StrictExitNodes* is true, then only nodes from that list can be chosen as exit nodes. So the configuration file can be changed to have a client always come to a friendly machine for an exit server.

(TS//SI) There is one other way to influence a client to choose a specific machine for the exit. Servers have exit policies, which are lists of types of requests that a server will handle. For example, by default, all servers reject traffic sent to port 25 (SMTP) in order to discourage spammers from using Tor's network to hide their activities. One can even set up a server that will reject all exit traffic, and only act as a first or second node. So if we set up a server with a more permissive exit policy than most other machines, we will be chosen more often to handle the sort of traffic that other machines will not allow. For example, if we were to allow connections to port 25, our server would be chosen to send e-mail with higher probability than it would for sending other types of traffic. We have no specific statistics regarding common exit policies, and even if we offer a certain exit policy, that doesn't mean that people will think to try to send such traffic through Tor. Still, this option is more attractive than altering a client configuration file, because it only requires a change on a machine that we own.

D.4 (U) Hidden service nodes

(U) As covered in appendix C, Tor uses special nodes when accessing hidden services.

D.4.1 (U) Introduction points

(U) The introduction point is the third node in a circuit that starts at the hidden service, so it knows the information listed under **Third Node** above. It is also, briefly, the third

node in a circuit coming from the client, but this circuit is shut down as soon as the intro point passes along the hidden service request. In addition, it knows the RSA public key (modulus) for the hidden service. This is more information than it needs, because when it receives a request to access the hidden service, all that is passed along is a SHA-1 hash of the public key, so all the introduction point really needs is that hash. By giving the introduction point the entire public key, Tor is leaking more information than it needs to. Of course, the only thing that the public key really gets us is the ability to connect to the hidden service, but that is not insignificant. Consider the (small) probability that a hidden service that we do not know about will select a friendly machine as one of its introduction points, and the usefulness is clear.

(S//SI) A machine knows that it is an introduction point as soon as it receives an ESTABLISH_INTRO cell, which contains the public key info.

(TS//SI) The hidden service can select its introduction points in its configuration file through a field called *HiddenServiceNodes*. By changing that field, we can influence the hidden service to choose our box as its introduction point, but if we are on their box already, there are far more useful things we can do, like obtaining the hidden service's private key. If *HiddenServiceNodes* is empty, then introduction points are picked using `router_choose_random_node`.

D.4.2 (U) Rendezvous points

(TS//SI) Rendezvous points are the most knowledgeable nodes in a hidden service circuit. When accessing a hidden service, a client must first set up a circuit to a rendezvous point (RP). The RP is the third node in this circuit, so it knows all the information a third node knows; see above. In addition to this basic third node information, the rendezvous point is also given a "rendezvous cookie", a random 20-byte number which the RP will use later. After this circuit is set up, the client requests access to the hidden service through the introduction point (the RP sees none of this), and if the hidden service decides to allow contact, it will set up a circuit to the RP. The RP is the third node in this circuit as well. The hidden service sends a cell down this circuit that contains the rendezvous cookie, g^y for a Diffie-Hellman handshake between the hidden service and the client, and K_H. Right now, the RP only looks at the rendezvous cookie, but it would require only a trivial change to have it store the other values.

(TS//SI) At the RP, after it decrypts its layer of encryption, the cell is only under one layer of encryption, based on the shared secret between the hidden service and the client. We can also see g^y. This is a far cry from knowing the shared secret, but this is as close to being able to see traffic between the client and hidden service as we can get right now. It is certainly the weakest point of the entire hidden service system.

(S//SI) A node knows that it is a rendezvous point as soon as it receives a RELAY_ESTABLISH_RENDEZVOUS cell, which contains the rendezvous cookie, and it knows that the hidden service and client are connected across it as soon as it receives a RELAY_RENDEZVOUS1 cell from the hidden service and sends a RELAY_RENDEZVOUS2 cell to the client.

(S//SI) Clients can be induced to choose a particular machine as a rendezvous point by changing the field *RendNodes* in their configuration file. If this field is empty, the RP will be chosen using `router_choose_random_node`. Hidden services will connect to whatever rendezvous point the client tells them to, even if that node is normally on their list of excluded nodes, assuming that it decides to communicate with the client.

E (U//FOUO) MJOLNIR API

(U//FOUO) MJOLNIR attempts to make accessing Tor functionality simple. Below is an API describing the most useful functions of MJOLNIR, arranged by topic.

E.1 (U) Initialization

- (U) `int tor_init (void)`
  - **Description:** This is the main entry point for the Tor client. It must be called before any Tor services are used.
  - **Arguments:** None.
  - **Return Value:** Return 0 on success; return a negative value on failure.
  - **Notes:** This *must* be called before any other Tor functions are used. It can be called multiple times without issue.
- (TS//SI) `int init_keys (const smartlist_t *subject, const smartlist_t *issuer)`
  - **Description:** In addition to important crypto initialization, `init_keys` generates X.509 certificates (through internal functions), which the program will use to communicate with the Tor cloud.
  - **Arguments:** The subject and issuer smartlists. See the documentation in the section on smartlists and certificates.
  - **Return Value:** Return 0 on success and negative values on failure.
  - **Notes:** This function must be called in order for the Tor client to have specific certificates. If no *commonName* field-value pair is passed in, it will generate a random *commonName* (10-20 characters long). See the documentation for subject and issuer smartlists.
- (TS//SI) `routerstatus_t *set_chosen_dirserver (const char *nickname, int fascistfirewall)`
  - **Description:** Call this with the nickname of any directory server, and then whenever MJOLNIR needs to fetch something, it will use that directory.
  - **Arguments:**

1. *nickname*: the nickname of the chosen directory server. If it is `NULL` or not the nickname for any valid server, the dirserver will be chosen randomly.
2. *fascistfirewall*: set to 1 if MJOLNIR is behind a firewall that will not let certain ports out. See Tor's man page for more documentation.

  - **Return Value:** The **routerstatus_t** structure describing the chosen directory server. The return value should be checked to see if it is `NULL` , indicating an error.
  - **Notes:** The nickname should be either a trusted directory server (hard-coded into the library) or a directory mirror. In order to download from a mirror, MJOLNIR must first know which servers are mirrors, so it will have to make at least one request from an authoritative directory.

- (TS//SI) `int update_router_descriptors (void)`
  - **Description:** Get the router descriptor data from a directory server.
  - **Arguments:** None.
  - **Return Value:** 0 on success; negative on failure.
  - **Notes:** This function must be called every time the program needs to update its listing of routers. This must be done at least once before setting up any circuits. It may be done an arbitrarily large number of times throughout the life of the program, but each fetch will generate a sizable amount of traffic to one directory server.

    Warning: this function will always talk to one of the 5 directory servers hard-coded into `config.c`:529 (`add_default_trusted_dirservers`). This behavior should be able to be changed if the chosen directory server is a directory mirror. See documentation for `set_chosen_dirserver`.

    Warning: This function is blocking. It will halt program execution until it either times out or receives a response. See the timeout documentation.

- (TS//SI) `char *get_routerdesc_string (void)`
  - **Description:** Return the exact string that the server returns when MJOLNIR asks for the directory.
  - **Arguments:** None.
  - **Return Value:** A new copy of a string containing all router descriptors as received from the directory.
  - **Notes:**

- (U) **routerlist_t**

The definition of `routerlist_t` is in `or.h` on lines 930-941. The most important field is the smartlist *routers* which is a smartlist of **routerinfo_t** structures. The one part

of these that will be used most often is probably the nickname field. Let us assume there is a routerlist named `rlist`. There are `rlist->routers->num_used` routers available, and to access their nicknames, use `rlist->routers->list[i]->nickname`. Also see the documentation of `SMARTLIST_FOREACH`.

- (TS//SI) **routerinfo_t**

  The function `router_get_routerlist` returns a smartlist of **routerinfo_t** structures (see above for more details). The most important fields of **routerinfo_t** are *nickname* and the *extend_info*. Using this library, the programmer should never have to deal directly with the internals of **extend_info_t**, and will probably never see it at all, but if the programmer want to add to the library, he will need to look over how it is used. The best place for a quick example is any of the attack functions detailed in `controlmod.c`.

- (U) `routerlist_t *router_get_routerlist(void)`
  - **Description:** Return the current list of all known routers.
  - **Arguments:** None.
  - **Return Value:** The list of all known routers. This list will not be populated until `update_router_descriptors` has run.
  - **Notes:** See the **routerlist_t** documentation.

- (TS//SI) `void tor_cleanup(void)`
  - **Description:** Cleans up before shutting Tor down.
  - **Arguments:** None.
  - **Return Value:** None.
  - **Notes:** This must *not* be called unless no more Tor connections or services are needed. Calling `tor_init` after calling this function does not work yet. It seems that the problem with this is actually in shutting down and resetting the OpenSSL library.

E.2 (S//SI) Circuit building

- (U) **circuit_t**

  The only field from this structure that the programmer needs to worry about is *global_identifier*. This identifier is how the program keeps track of circuits it has built, so if a program wants to have multiple circuits up at once, the programmer must keep track of their respective global identifiers or maintain a list of pointers to the actual structures.

- (TS//SI) `circuit_t *new_arbitrary_circuit_by_nickname (char * begin_router_nickname)`

- **Description:** Begin a new circuit that can be added to at any time so long as the following two conditions are met:
  1. The client is not one of the nodes in the circuit.
  2. The same node is not used twice in a row. (It can, however, be used twice in the same circuit.)

  Note that the first of these is not checked in code, but in its current state, a client using this library will never be a server, so this will not be an issue.
- **Arguments:** *begin_router_nickname*: The nickname of the first hop in the new circuit.
- **Return Value:** A pointer to the new circuit.
- **Notes:** This function must be called before attempting to append a new hop onto a circuit.

  Warning: This call is blocking. It will halt program execution until a circuit has been successfully established.

• (TS//SI) `int append_router_by_nickname_to_circuit_by_id (char *nickname, int circ_id)`

- **Description:** This function appends the router with the given nickname to the given circuit. It looks up the circuit by its global identifier.
- **Arguments:**
  1. *nickname*: The nickname of the next hop in the circuit.
  2. *circ_id*: The global identifier of the circuit.
- **Return Value:** 0 on success; negative values on failure.
- **Notes:** Must get the circuit id from a circuit returned by `new_arbitrary_circuit_by_nickname`. If MJOLNIR tries to connect to an invalid router (e.g. if the router is down), the circuit will close.

  Warning: This call is blocking. It will halt program execution until a circuit has been successfully extended.

• (TS//SI) `int append_router_by_nickname_to_circuit (char *nickname, circuit_t *circ)`

- **Description:** Append router with the given nickname to the given circuit.
- **Arguments:**
  1. *nickname*: The nickname of the next hop in the circuit.
  2. *circ*: The circuit to which MJOLNIR should append a router.
- **Return Value:** 0 on success; negative values on failure.

- **Notes:** Must get the circuit from `new_arbitrary_circuit_by_nickname`.
  Warning: This call is blocking. It will halt program execution until a circuit has been successfully extended.

- (TS//SI) `routerinfo_t *get_random_router (routerinfo_t *exclude)`
  - **Description:** This function gets a random router from the global router list so long as the chosen router is not the excluded one.
  - **Arguments:** *exclude*: Gives the programmer the option to exclude a certain router. It is most useful to exclude the most recently appended router in the circuit, as the circuit dies if it tries to go to the same router twice in a row.
  - **Return Value:** Returns the **routerinfo_t** for the chosen router. This is probably only useful for extending the current attacks, in which case it would be useful to study those attacks carefully.
  - **Notes:**

- (TS//SI) `int nickname_is_known (const char *nickname)`
  - **Description:** This function checks to see that the nickname is a valid one.
  - **Arguments:** *nickname*: a string with the nickname of some router.
  - **Return Value:** Returns 1 if the router is valid; returns 0 otherwise. The router is valid if the program has **routerstatus_t** information about the router with that nickname.
  - **Notes:** The program may know about an invalid router. For example, the router may have been taken down since the router descriptor was fetched. However, when MJOLNIR tries to connect a circuit, it will fail.

- (TS//SI) `void close_circuit(circuit_t **circ)`
  - **Description:** Given an open circuit, close it, remove it from *global_circuitlist*, and de-allocate its memory. *circ* is `NULL` on return.
  - **Arguments:** *circ*: The circuit to be closed.
  - **Return Value:** None. However, *circ* is `NULL` so it can no longer be used.
  - **Notes:**

E.3 (U//FOUO) Sending and receiving data

- (TS//SI) `int send_payload_down_circ(char *payload, size_t payload_len char *ip_addr, int port, circuit_t *circ, int force)`
  - **Description:** This function sends a custom packet down the requested circuit to the specified address at the specified port.

- **Arguments:**
  1. *payload*: the custom payload, e.g. an `HTTP GET` request.
  2. *payload_len*: length of the payload.
  3. *ip_addr*: the IP address of the ultimate destination in IPv4 form: 192.168.1.1 (as a string).
  4. *port*: the destination port.
  5. *circ*: the circuit across which to send this packet.
  6. *force*: if true, the function will always try to send a payload. If false, it will check that the address is reachable with the current exit, and if it is not, it will not send a payload.
- **Return Value:** Return 0 on success; negative values on failure. If it refuses to send a packet because it was not forced, it returns failure.
- **Notes:** It is up to the calling program to check that both the payload and IP address:port are valid before calling this function. The *payload_read_handler* must be set before this call is made.

  Warning: This call will block program execution until it times out or receives a response.

• (TS//SI) *payload_read_handler* (function pointer):
`void * (*payload_read_handler) (const char *, int)`

- **Description:** Function pointer to a user-defined function that gets called when the client receives a payload in response to its request.
- **Arguments:**
  1. `const char *`: the payload that the client receives in response to its request (what the calling program actually wants to read).
  2. `int`: length of the payload.
- **Return Value:** Must return a `void *`, so the return value can be anything the programmer wants.
- **Notes:** These functions should be defined by the user. To set this variable, call `set_payload_read_handler (handler)`. Example functions (`print_payload` and `do_nothing`) are given in `torm.c`.

  (TS//SI) **Timeouts.** MJOLNIR times out if it does not receive a response in a certain amount of time. On the test LAN, we have it set to .1 s, which is more than enough time. However, due to the slow nature of the Tor cloud, it may be necessary to change the value of the initial timeout estimate. Our timeout follows the TCP-style timeout rules, so once it has started receiving cells, it should adjust the timeout on its own. The constants that may need adjusting are in `timeout.c`. In particular, look at `INITIAL_EST_SEC` and `INITIAL_EST_USEC` for the number of seconds and microseconds, respectively, before it times out. See the discussion in the section on sending and receiving data.

- (TS//SI) `int router_should_reach_addr (int addr, int port, routerinfo_t *router)`
  - **Description:** The function checks the router's exit policy and returns non-zero if the address is reachable and zero if the address is not reachable.
  - **Arguments:**
    1. *addr*: the address to reach, as an integer.
    2. *port*: the port to use at *addr*.
    3. *router*: the router to check.
  - **Return Value:** If *addr:port* is reachable, returns 1. If not, returns 0.
  - **Notes:**
- (TS//SI) `int router_should_reach_addr_string (char *ip_addr, int port, routerinfo_t *router)`
  - **Description:** This function checks the router's exit policy and returns non-zero if the address is reachable and zero if the address is not reachable.
  - **Arguments:**
    1. *ip_addr*: the address to reach, as a string in xxx.xxx.xxx.xxx form.
    2. *port*: the port to use at *ip_addr*.
    3. *router*: the router to check.
  - **Return Value:** If the *ip_addr:port* is reachable, it returns 1. If not, it returns 0.
  - **Notes:** This function just converts the string address to an integer and calls `router_should_reach_addr`.
- (TS//SI) `int circuit_should_reach_addr (int addr, int port, circuit_t *circ)`
  - **Description:** MJOLNIR checks the exit policy of the circuit's exit hop and returns non-zero if the address is reachable or zero if the address is unreachable.
  - **Arguments:**
    1. *addr*: the address to reach, as an integer.
    2. *port*: the port to use at *addr*.
    3. *circ*: the circuit want to check.
  - **Return Value:** If the *addr:port* is reachable, returns 1. If not, returns 0.
  - **Notes:** This function just grabs the last hop in the circuit calls `router_should_reach_addr`.
- (TS//SI) `int circuit_should_reach_addr_string (char *ip_addr, int port, circuit_t *circ)`

- **Description:** MJOLNIR checks the exit policy of the circuit's exit hop and returns non-zero if the address is reachable or zero if the address is unreachable.
- **Arguments:**
  1. *ip_addr*: the address to reach, as a string in xxx.xxx.xxx.xxx form.
  2. *port*: the port to use at *ip_addr*.
  3. *circ*: the circuit to check.
- **Return Value:** If *ip_addr:port* is reachable, returns 1. If not, returns 0.
- **Notes:** This function just converts the string address to an integer and calls `circuit_should_reach_addr`.

E.4 (S//SI) Smartlists and certificate masking

- (U) **smartlist_t**

  (TS//SI) The **smartlist_t** structure contains an array of `void *`s, the number of elements used, and its total capacity. However, the programmer should never have to deal with the internals directly. Instead, he should use `smartlist_create`, `smartlist_add`, and `SMARTLIST_FOREACH`. We use this to define custom X.509 certificates; it is nearly ubiquitous in the rest of the Tor code.

- (U) `smartlist_t *smartlist_create (void)`
  - **Description:** Allocate and return an empty smartlist.
  - **Arguments:** None.
  - **Return Value:** A pointer to the new smartlist.
  - **Notes:** Any smartlist the programmer wants to use must first be returned by this function.

- (U) `void smartlist_add (smartlist_t *sl, void *element)`
  - **Description:** Appends an element to the end of the list.
  - **Arguments:** *sl* is the smartlist that is receiving the element. It must have been returned by `smartlist_create`.
  - **Return Value:** None.
  - **Notes:** Technically, a smartlist may contain pointers to different kinds of objects, though this is highly unrecommended. Individual elements of the smartlist may be accessed through its list, e.g. `sl->list[i]`.

- (U) `void smartlist_remove (smartlist_t *sl, void *element)`
  - **Description:** Removes the element from the list.

  - **Arguments:** *sl* is the smartlist that is losing the element. *sl* must have been returned by `smartlist_create`.
  - **Return Value:** None.
  - **Notes:** Preserves the order of any elements before *element*, but elements after it may be rearranged.

- (TS//SI) `SMARTLIST_FOREACH (smartlist, type, var, code)`
  - **Description:** `SMARTLIST_FOREACH` is a macro that will run code for every element in a smartlist. All elements of the smartlist must be of the given type.
  - **Arguments:**
    1. *smartlist*: The smartlist over which MJOLNIR runs the code.
    2. *type*: The type of data held by the smartlist.
    3. *var*: The variable name by which the macro access the elements of the smartlist. The variable does not need to be declared before being used.
    4. *code*: The code to be executed. If there is more than one statement, it should be in curly braces.
  - **Return Value:** None.
  - **Notes:** Remember that this is a macro. If the code is bracketed, it must be followed the closing brace with a right parenthesis and semicolon.

- (TS//SI) **field_value_pair_t**

  (TS//SI) This is a very simple structure containing two strings: a field and a value. These should be allocated and set as any other strings. In the example program, this structure is added to a smartlist so that MJOLNIR can use custom X.509 certificates.

- (TS//SI) Subject and issuer smartlists (**smartlist_t**)

  (TS//SI) Tor uses self-signed X.509 certificates during its TLS handshake. By default, it sends only *commonName* (normally the client or server's nickname) and *organizationName* (normally the string "TOR") for both subject and issuer, with the only difference being that the issuer common name is followed by the string '<identity>'.

  (TS//SI) A Tor server will check for required X.509 fields, but not anything else except the subject's *commonName*, and the only restrictions are that the length is greater than 0 and it consists of alphanumeric characters. (The *commonName* can also contain a period, but this creates a log message on the other server. Currently, MJOLNIR treats the period as an illegal character.) These lists allow the programmer to send any X.509 certificate he wishes, instead of always sending the same ones. If there is no *commonName* in the subject section, the client will generate a random string of characters 10-20 bytes long to act as the *commonName*.

(TS//SI) If the *commonName* has an illegal character, MJOLNIR will generate a random one. If MJOLNIR receives as input an invalid X.509 field, it will ignore that field/value pair entirely. In this way, MJOLNIR provides relatively robust X.509 certificate generation.

(TS//SI) The programmer must create two smartlists of **field_value_pair_t** structures (one each for subject and issuer) where the field variable is the name of the field in the certificate and value is set to the appropriate string. These two smartlists must then be passed to `init_keys` (subject first, issuer second).

(TS//SI) See the documentation for **smartlist_t**, **field_value_pair_t**, `smartlist_add`, `smartlist_create`, and `init_keys`.

E.5 (TS//SI) DoS-style attacks

(TS//SI) The purpose of these attacks is not actual DoS *per se*. These simply do not generate enough traffic. However, they may be able to generate sufficient traffic to force a node into hibernation, which takes the server out of the Tor cloud until the next hibernation period begins (could be a day, week, or month).

- (TS//SI) `circuit_t *flower_attack_one_router (char *routernick, int num_petals)`
  - **Description:** Set up a (D)DoS network attack on one Tor node. The program will create a circuit consisting of random routers alternating with the given router. This results in the given router doing much more work than the rest of the nodes on the circuit.
  - **Arguments:**
    1. *routernick*: the nickname of the router to attack.
    2. *num_petals*: he number of times the attacked router will be in the resultant circuit. Therefore, the circuit will be 2 * *num_petals* long.
  - **Return Value:** A pointer to the **circuit_t** structure of the newly-formed circuit.
  - **Notes:** When setting up a circuit with n hops, the client must send n cells along the circuit. So setting up a large circuit can be very time-consuming, as well as expensive with regards to both traffic generated and CPU time used.

    Warning: This call is blocking. It will halt program execution until a circuit has been successfully established. For even a moderately long circuit, this can take a substantial amount of time.

- (TS//SI) `circuit_t *append_flower_to_circuit (char *routernick1, int num_petals, circuit_t *circ)`
  - **Description:** Similar to `append_coil_to_circuit`, this function appends a flower to the end of the given circuit. Using this, MJOLNIR is anonymized by the structure of the Tor cloud.

- **Arguments:**
  1. *routernick1*: The router in the middle of the flower.
  2. *num_petals*: The number of times the middle router appears in the flower. So the flower will be 2 * `num_petals` long.
  3. *circ*: The original circuit to append the new flower to.
- **Return Value:** The circuit with the flower added.
- **Notes:** See the flower attack documentation.

• (TS//SI) `circuit_t *coil_attack_two_routers(char *routernick1, char *routernick2, int num_coils)`

- **Description:** Similar to the flower attack, except that this function generates a circuit that bounces back and forth between two nodes in the Tor cloud instead of targeting only one router.
- **Arguments:**
  1. *routernick1*: nickname of the first attacked router.
  2. *routernick2*: nickname of the second attacked router; if MJOLNIR sends a payload down this circuit, the second router will be the final one.
  3. *num_coils*: the number of times each router will appear in the circuit. So the circuit is actually 2 * `num_coils` long.
- **Return Value:** A pointer to the newly-formed circuit.
- **Notes:** See the notes under `flower_attack_one_router`; the same applies here.

  Warning: This call is blocking. It will halt program execution until a circuit has been successfully established.

• (TS//SI) `circuit_t *append_coil_to_circuit (char *routernick1, char *routernick2, int num_coils, circuit_t *circ)`

- **Description:** Similar to the coil attack, but puts the coil at the end of a given circuit. Using this, MJOLNIR is anonymized by the structure of the Tor cloud.
- **Arguments:**
  1. *routernick1*: the first router in the coil.
  2. *routernick2*: the second router in the coil.
  3. *num_coils*: The number of times each router appears in the coil. The resulting circuit will be as long as `circ` plus 2 * `num_coils`.
  4. *circ*: The circuit to which MJOLNIR appends the coil.
- **Return Value:** the newly-formed circuit.

– **Notes:** See the documentation for the coil attack.

  Warning: This call is blocking. It will halt program execution until a circuit has been successfully extended.

F (U//FOUO) MJOLNIR for Windows

F.1 (U) Interface

(TS//SI) The GUI provides the user with a list of all the circuits created by the client. The list displays relevant information about each circuit, including its length, state, and purpose. Additionally, each circuit has an light icon next to its name. The light is red if the circuit is closed (cannot be used). The light is green if the circuit is open and ready for use. Beneath the circuit list, a list of the servers in a circuit is displayed for the last circuit clicked. Similar to the circuits, the servers each have a light next to their names to indicate whether the circuit is open up to that server. The displays for the circuit list and the node list are set to update on a timer (currently set to three seconds.) On the right, the GUI provides a list of servers that are available for use in a circuit. The router list does not update on the timer. To update the router list, the user must click the "Load Routers" button.

F.2 (S//SI) Creating new circuits

(TS//SI) The GUI provides a few different ways to create new circuits. The most general way to create a new circuit is to click the "New Circuit" button. This will create a null circuit with a user-specified name and purpose (see the Tor documentation for more information on circuit purpose.) The user can then proceed to build a circuit by appending servers individually to the end of the circuit. Alternatively, the user can create a random circuit of arbitrary length by clicking the "New Random Circuit" button. The user then specifies a name and length n, and MJOLNIR will choose n random servers from the available router list and create a new circuit. Currently, all random circuits are made with "General" purpose. Modifying the program so that the user can specify the purpose of a random circuit only requires a minor change at the library level. However, it is likely that creating circuits of certain purposes will cause the program to crash (see below.) Circuits can also be created using the coil or flower attacks (see section 7.1, **DoS-Style Attacks**.)

(TS//SI) To append a server to a circuit, the user must first select the desired circuit by clicking on its name in the circuit list. This causes the chosen circuit to become current, which means it is the circuit whose node list is currently displayed. Then the user right-clicks on the name of the desired server in the router list and selects "Append to Current Circuit." Any "General" purpose circuit that is open can have a server appended to it.

(TS//SI) It should be noted that circuits created with certain purposes cannot have servers appended to them, and trying to do so will crash the program. Currently, only "General", "Introducing", and "Establish Rend" circuits are able to have servers appended

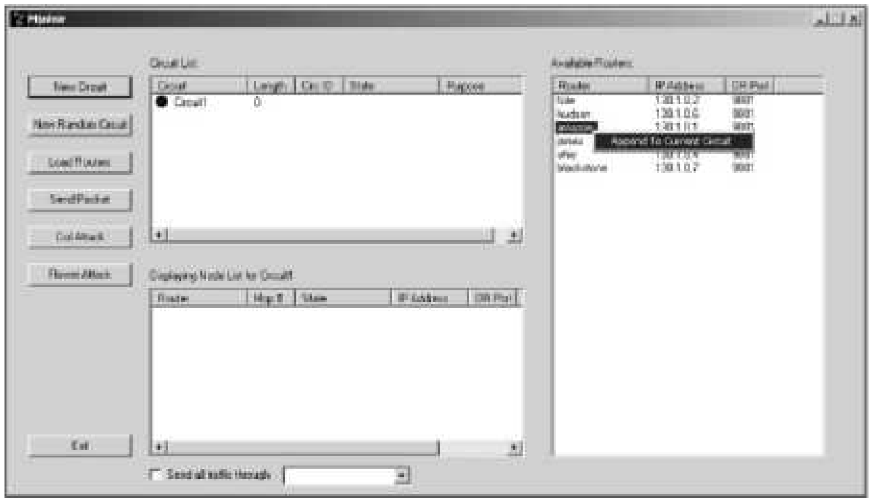


Figure 3: **(TS//SI) Appending a server to a pre-existing circuit.**

to them. Once a server is appended to an "Establish Rend" circuit, its purpose becomes "Rend Ready" (a one-hop circuit to a rendezvous point) and attempting to append any more servers will cause a crash. This will need to be cleaned up as functionality for creating rendezvous circuits and accessing hidden services is added.

F.3 (TS//SI) DoS-style attacks

(TS//SI) The GUI provides an interface for performing the coil and flower DoS attacks from the MJOLNIR library. In the coil attack, the user chooses two servers to attack and the number of coils in the circuit. Each server appears once in the circuit for each coil, so the length of the circuit will be twice the number of coils. For the flower attack, the user chooses a server to attack and the number of petals, where the number of petals is the number of times the selected server will appear in the circuit.

(TS//SI) For both attacks, the user has the option to create the coil or flower as a new circuit or to append it to an existing circuit. However, appending coils or flowers is more expensive than creating them as new circuits, and there is a significant performance hit for appending large coils and flowers to circuits.

F.4 (TS//SI) Sending arbitrary packets

(TS//SI) To send a packet through a circuit, the user first clicks the "Send Packet" button. Then the user selects a circuit to send the message through, and an IP address and port number to send the message to. The user has the option of entering the payload of the packet in either hex or ASCII. Upon clicking "Send", the packet is encrypted with the appropriate layers of encryption and sent through the circuit. The exit server will then send the decrypted packet to the selected IP address and destination port.

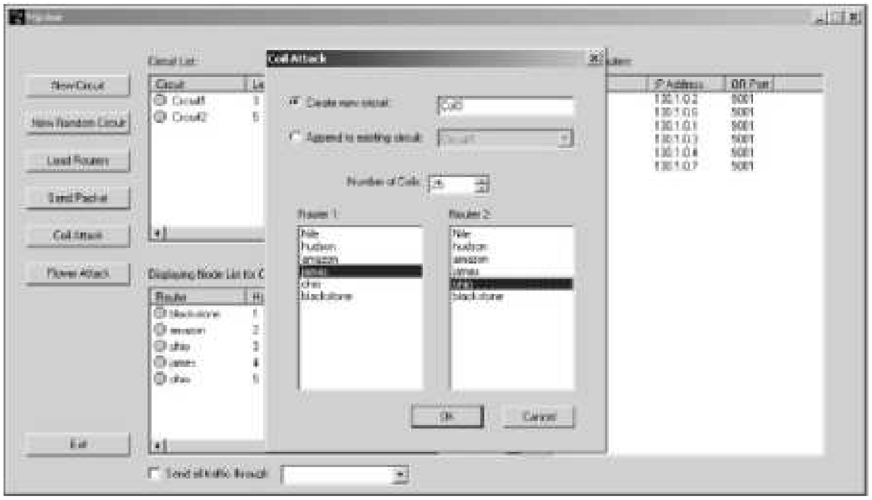


Figure 4: **(TS//SI) The coil attack dialog.**

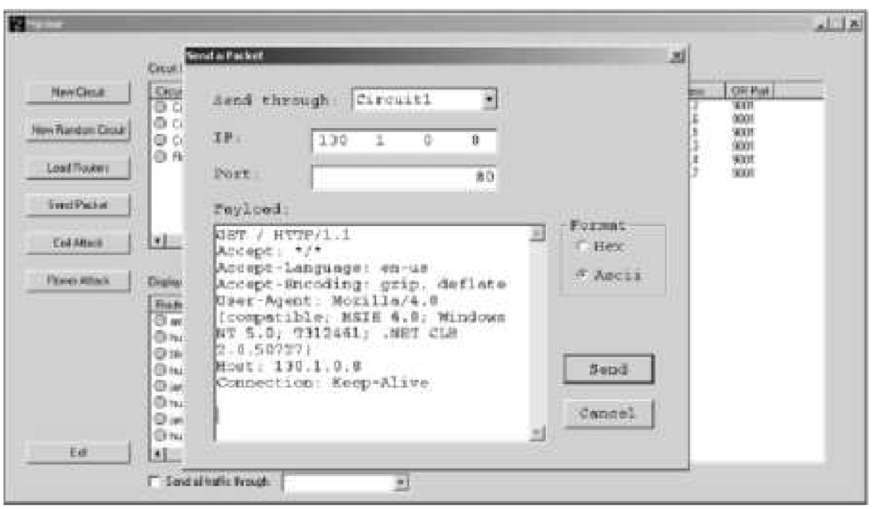


Figure 5: **(TS//SI) Sending a custom TCP packet using the MJOLNIR GUI.**

F.5 (U) Future features

- Customizing the certificates. This feature is currently available in the MJOLNIR library, but needs an interface in the GUI.
- Functionality for creating rendezvous circuits and accessing hidden services as it becomes available in the library.

F.5.1 (U) Minor changes/bug fixes

- Add a purpose field to the "New Random Circuit" dialog box, and deal with issues that arise from attempting to create circuits of certain purposes.
- Address the crashes caused by appending servers to circuits of certain purposes either by refusing to allow such an action or fixing it at the library level.